Personal Service Environments – Openness and User Control in User-Service Interaction

Markus Bylund and Annika Waern

Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, Sweden
{bylund, annika}@sics.se

Abstract. We describe an approach for mobile and personalized use of electronic services that meet very high requirements on openness, user control, and mobility. The design is centered on the concept of personal service environments. These offer users mobile and network independent access to services from many different types of devices. The concept also allows services to interact locally. This can be used in several ways: services can for example share information but also make use of each other's services. In particular, we want to use this to support service personalization. The design has been successfully implemented and tested with a number of sample services and in several related research projects. This implementation, the *s*View system, is described.

Keywords. Electronic services, personal service environments, user control, ubiquitous computing, user interfaces, mobility, personalization, service collaboration.

May 2001
SICS Technical Report T2001/07
ISSN 1100-3154
ISRN: SICS-T--2001/07-SE

1. Introduction

Today, most people in the Western world are used to using Internet-based services. We use services – almost exclusively placed in the World Wide Web (WWW) framework – to search for information, to buy books, and to book airline tickets. We use them to communicate with our friends, and to make new friends. A home computer is no longer made useful only through the stack of CDs on our bookshelf – it is a point of access to a vast and perpetually changing world of entertainment, shopping, business and general information resources. And as if this was not enough, we are rapidly acquiring other devices for access to the Internet world of services: mobile phones, communicators, home gateways and Internet-equipped game consoles.

This rapid development has created a mismatch between the development of Internet content and the technology for content provision. Content is developing into a set of independent (but potentially combinable), device-independent, and personalizable services. But the predominant technology still provides a strongly server-based functionality, where services run entirely at the server side, accessed through media-specific, content-insensitive, interaction devices (HTML/WML browsers, media players, etc.) with little support for user personalization.

We see two main problems with this development. Firstly, current technology lacks true support for nomadism and continuous interaction channels through multiple devices [1, 2]. Secondly, technology put services almost entirely in the control of service providers. This gives rise to privacy problems, such as the fact that service providers have complete control over the information users provide to personal services. Even more importantly, it makes it difficult to combine services to support specific users' need. For example, there are still very few Web sites that offer automatic comparisons between sales offers from other Web sites, despite the clear benefit of such services.

These issues have not been left unaddressed in research. In particular, work on mobile agent technology and multi-agent systems (see Section 4.2) has aimed to address the issues of service nomadism and interaction. This paper takes a similar but slightly different perspective: we explore how Internet-based services, rather than generic software agents, can be developed to be nomadic, personalizable, and provide possibilities for service interaction.

The solution proposed is that of a *personal service environment*. A personal service environment is an individually collected and tailored set of services, available to the user at all times, and at least partially independent of Internet access. The services are retrieved from service providers around the Internet, and the personal service environment itself is mobile, following its user around in the network.

We have designed and developed a Java-based system for electronic services that is based on the notion of personal service environments – *sView*. In this system, personal service environments are composed both of services that are mobile and follow the user, and of services that are platform or location specific. In this way, *s*View provides personal service environments that are tailored both to the user and to

the usage context. The design of *s*View is highly modular. In fact, it could replace the Web browser as such, as some of the services may well be provided to support presentation and interaction.

The paper is structured as follows. Section 2 is a requirements analysis; in this Section we analyze the requirements on openness and user control in more depth. Section 3 presents personal service environments and shows how this concept has been designed to meet the requirements from Section 2. Section 4 covers existing Web technology and some alternative approaches in terms of how well they are able to fulfill our requirements. Finally, Section 5 presents the *s*View system, and Section 6 describes a few experiences that build on the work presented in this paper.

2. Requirements Analysis

We see two requirements as central for an infrastructure for electronic services. The first is that it must be *open*. It should be possible to add and remove services and users without affecting other services or users. The second requirement is that it must be *controlled by the user*. An infrastructure for electronic services should give the user control over which services to use, what information about the user that services handle, how services collaborate, etc. Some users may not ever do so, but the possibility for user control should always exist. Furthermore, the user should be in control of the usage situation. In practice this means that services should be reachable from everywhere using many different types of devices, both the user's own devices and publicly available devices.

These requirements on openness and user control imply a number of more specific requirements, which we now will discuss in more detail.

2.1. Heterogeneity

Many electronic services already exist, both in the form of commercial and research products. A sound requirement on an open infrastructure for user-service interaction is to allow a heterogeneous mix of service components to utilize features of each other.

2.2. Extendibility

Openness also implies a demand for extendibility. As new services are added to the system it should be possible to add support for new protocols for user-service interaction, protocols for communication between service components, support for collaboration between services of different kinds, protocols and algorithms for implementing security functionality, and more.

2.3. Accessibility

User control requires that all users always can access the service infrastructure. Users should be able to access services while on the move, not just from the office or from home, but also while riding the bus, in an airplane, on the street while shopping, etc. Disabled and elderly users must be able to access the infrastructure, as well as children.

2.4. Adaptability

Accessibility is only the minimal requirement for user control, but it poses already very high demands on adaptability. The service infrastructure must be extendable and adaptable to a wide range of input devices [3]. Another requirement is network adaptability: the infrastructure, and services, must be able to adapt to variations in network connectivity and bandwidth [4, 5].

Services should also be able to adapt to their users, or rather the preferences, experiences, or usage history of their users. This is a delicate issue since adapting to qualities of individual users requires services to handle personal information, which in turn may jeopardize the privacy of users [6]. The requirement on user control means that personalization must be done in a way that ensures user privacy. Furthermore, the infrastructure should ensure that the task of managing personalization does not overburden the user.

2.5. Continuity

Finally, user control requires that services not only are accessible from multiple devices, but also maintain their state when the user switches between devices. Users should not need to restart a service just because they move over to another device. When switching between devices, a user should be able to resume his or her interaction with a service exactly where it was suspended.

3. Proposed Approach

The personal service environment concept describes a service infrastructure that is targeted to fulfill the requirements from Section 2. It is a runtime environment that is private to an individual user, and functions as a briefcase for his or her electronic services. As with the Web, we assume that services themselves use a client-server model. In contrast to the Web however, services can store both logic and data locally within the personal service environment, and there is no predefined split between what should be performed on the client and on the server sides.

The service environment infrastructure fulfills our requirements on openness. Any individual or organization with an Internet connection can own an environment in which services can be stored and executed. In the same way, anyone can publish

services for use in an arbitrary service environment. Adding a new user or service to the system does not affect already present users or services.

The most important requirement we pose on service environments is that the environment itself must be mobile. It should be able to follow its user in the network between e.g. a workstation when the user is in the office, to a notebook computer when the user is on the road, or to a shared server for service environments when the user lacks immediate access to the network. As the environment migrates, the services it stores should follow, and the state of the services and their ways of interacting with each other should be preserved. We also require that the service environment can move between client devices without loss of interaction state.

There are several reasons for making the service environment mobile and execute services locally. Firstly, a service that executes locally is not necessarily dependent on a network connection. Secondly, a local service is likely to have access to richer user interface types than remote interfaces. Thin devices with Internet access, and possibly with a less powerful interface (e.g. WAP/WML capable phones), can be used to access the environment on a networked host. Finally, by having parts of the functionality of services executing on users clients, the total CPU processing of a service is distributed between the users, which alleviates base services.

A key feature with the personal service environment is that it provides a natural boundary for service-service interaction. Services within an environment could be allowed to publish their APIs (Application Programming Interfaces) to each other.

To meet requirements on user accessibility, personal service environments must also support numerous channels for user interaction, e.g. HTML over HTTP, WML over WAP, Graphical User Interfaces (GUI), etc. The architecture must be open to enable integration of novel interaction models over time.

Since service environments are personal and follow their users around, they provide an ideal place for storing personal information for use by services (e.g. preferences and contact information). Whenever a user wants to add or change his or her personal information, or just inspect the information, it can be done in one place for all services. Service providers get a central access point to personal information of each user, information that can be shared with other services (with the user's permission).

3.1. A Usage Scenario

Below we present a usage scenario that illustrates the use of personal service environments and a few services.

> *A man is about to make a business trip to Cairo. Using his personal service environment search tool on his desktop computer he locates a travel agency service and initiates a dialog with it.*
>
> *The travel agency uploads a travel service component to the user's service environment.*
>
> *Once in the service environment, the travel service receives the man's instructions, via a standard graphical user interface (GUI), to make a flight and hotel reservation for his planned trip.*

Then the man turns his attention to something else and leaves the office. But before doing so, he lets his service environment know that he is no longer available via his desktop computer but rather via his cellular phone.

The travel service now makes use of a number of information sources in order to accomplish its task. It searches the service environment for a preference manager and asks it about its client's complete name and address, as well as his seating and smoking preferences. It also locates a calendar within the service environment and checks when the man must be back and if the trip conflicts with any of his other appointments.

Having collected all background information, the travel service turns to its base service trying to find an appropriate flight and hotel. The service finds three alternatives that all match the man's request, preferences, and schedule.

The travel agency is now ready to get back to the client with the result of the search. However, since the man is no longer available via the desktop computer, the service contacts him via his cellular phone. The man, now on the train on his way home, selects one of the alternatives and instructs the travel agency to go ahead with the reservation.

The travel service accepts the request and starts searching the client's service environment again, this time for a service that provides payment. One of the man's services, a bank service, is willing to provide payment, but only after a confirmation by the user (this is also done through the interface of the cellular telephone).

Having everything that is needed, including payment, the travel service now executes the man's request by instructing its base service to buy the flight tickets and make the hotel reservation.

4. Related Technologies

There exist many techniques and systems that fulfill some of the requirements of Section 2. The concept of personal service environments has in many ways been inspired by existing work on WWW enhancements, as well as of experiences with mobile agent technologies. There are also examples of more recent technology that fulfils some, but not all, of our requirements.

4.1. The World Wide Web

The WWW was originally intended to combine hypertext and text retrieval to get a "global information universe into existence using available technology" [7]. Considering these design goals, it is truly remarkable that the Web has been able to take on the role as an infrastructure for general user-service interaction as it plays today, with demands on highly interactive interfaces, mobility, and personalization. The ambitions of the inventors of the WWW to make the Web extendible, platform independent, and transparent has clearly played a key-role in this development. However, with the requirements in Section 2 in mind, the WWW is facing some challenges.

Services mediated through the WWW require a Web browser for user interaction. Web browsers in turn, most often require a quite powerful computer as host, as well as a keyboard and a mouse, in order to function. This limits the types of devices that users can reach services from. Telephones (both traditional and cellular), palmtops, and other special purpose devices can be used in some cases, but only by extending the WWW with separate interaction systems such as WAP/WML technology.

On the WWW, service logic and data are hosted by the set of backend services used by an individual user. This results in a dependency on the network connection between users and services; if the connection fails, not even the simplest functionality of a service can be utilized. In many cases (e.g. bank services), the scheme used to ensure privacy makes it impossible to even view information that was viewed only moments ago, just before the connection broke.

The support for saving the state of the user-service interaction is also limited. If a user is in the middle of a session with a service the user cannot suspend the interaction in order to resume it from someplace else. This is because WWW clients, through HTTP, are stateless [7]. The state of the user's services is instead distributed across all of the user's service providers, which makes it difficult to find a general solution to the problem.

There is little support built into Web browsers for personalizing services. Essentially, the only way to handle it is by having the service provider identify the user in order to tailor the interaction at the back-end of the service. The problem is worsened by the fact that personal information of individual users is distributed across all of their service providers. As the number of services in use increases, the user soon loses control over the personalization process. Also, all information that is needed for personalization, no matter how sensitive to the user, needs to be passed to the service via a network. This opens for privacy issues.

While it is a strength of the WWW that no information about other services is needed in order to add a new one, the lack of a general way to obtain information about other services makes service collaboration difficult. This is a two-faced problem. Firstly, services have difficulties finding peers to collaborate with since there is no uniform way for services to publish their capabilities to other services. Secondly, how do they actually collaborate once a peer is found? The APIs of Web-based services are typically made for humans using protocols that are very awkward for machine-based services to utilize. While it is relatively easy for users to find and use such services, these problems make it difficult for end-users to combine the services' functionality. The Simple Object Access Protocol is an example of a recent initiative to relieve the latter problem with the WWW [8].

General Web Extensions. Web organizers (e.g. www.eorganizer.com) and virtual desktops (e.g. www.magicaldesk.com) provide their users with integrated suites of Web-based e-mail handling, calendar, on-line storage of data, and sometimes news and games as well. In some cases, the services go as far as to simulate a desktop environment of an ordinary personal computer, complete with folders, desktop icons and even drag-and-drop functionality. In a way, this approach is similar to personal service environments. Even closer comes NetChaser [9], which is a system that supports personal mobility of Internet services such as the WWW, FTP, and e-mail. The system offers its user a personal view of his or her services via WWW browsers.

The system keeps track of the state of its user's services, which makes it possible for a user to start a session on one WWW client, suspend the session, and resume it again from a different client.

The major difference between these approaches and the personal service environment concept is that the former do not meet our requirements on openness. Web organizers are not in any way open for everyone to add new services. Furthermore, since these services rely on Web technology, they are not able to adapt to changing bandwidth availability or intermittent Internet access.

4.2. Mobile Agent Environments

Personal service environments bears many similarities to general Mobile Agent Environments (MAE) [10] and the concept was partly inspired by experiences from projects in which MAE were applied [11, 12]. They both provide environments that support dynamic loading of lightweight software components, as well as migration between such environments.

Many application examples apply mobile agent technology to meet requirements that are similar to those analyzed in Section 2. For example, Minar et al. [13] use mobile agents as a primary abstraction for creating dynamic and distributed systems with a focus on embedded computers. Hive agents are self-describing, mobile and capable of dynamic collaboration. Users are given a high degree of control in that they can create and manipulate (kill and move) agents using a GUI. The user can also create new applications by connecting agents with different capabilities, just by drawing lines between them. The system described by Minar et al. can be seen as a potential high-end interface for personal service environments.

Pullela et al. [14] describe a middleware that dynamically distributes computations in a mobile environment. The distribution is based on what resources are currently available at the mobile client. In this, Pullela et al. fulfill our requirement on access from multiple platforms, including very thin clients. They make use of the Ronin Agent Framework [15] that mixes agent-oriented and service-oriented paradigms for creating dynamic distributed systems. The Ronin Agent Framework shares the requirement on heterogeneity with the PSE concept, and meets it by including a meta agent communication language and a network independent agent communication message mechanism.

The Nomad system [16] is an advanced example of a service built on mobile agent technology. It allows mobile agents to travel to an auction service provider and participate in auctions on their user's behalf. Agents can place bids, learn and collect information, and set up new auctions. The Nomad system is an example of a type of service that goes beyond the requirements posed in Section 2, since the agents are initiated by users and travel to servers.

While MAEs have been a great source of inspiration when designing the personal service environment, the two are not altogether the same. A personal service environment is, in contrast to MAEs in general, specialized towards a particular task: to enable user interaction with networked services. This means that much of the functionality that is traditionally associated with MAEs can be simplified or removed [12]. For example, client side service components need not be able to migrate freely

between any two service environments; it is enough if a service component can be created at its base service and then moved to its owner's environment. Assumptions can also be made about the scope of a service component. It only needs to know about its base service and the other service components within the same environment.

Service environments on the other hand pose higher, or at least more complex, requirements on mobility and persistence than pure MAEs. A personal service environment must support persistence of itself as well as the service components that it houses. It also controls how services are allowed to move. Once services have been initialized in a service environment, they do not move individually over the network. Their mobility is rather controlled by the service environment they belong to.

In summary, although personal service environments could be implemented using MAE technology, they are not subsumed by it. As an implementation option for personal service environments, mobile agents introduce unnecessary overhead, as many central functionalities would be used only to a limited extent.

4.3. The OSGi Service Gateway

OSGi (the Open Services Gateway Initiative) provides a specification of an open framework for a service gateway [17]. The gateway can be loaded with multiple software services and it can execute on a number of platforms. The goal with OSGi service gateway is to create a common programming model for consumer services in which implementations are separated from their functional descriptions, and to create a simple and self-contained format for distribution of services. The former goal allows consumers to make use of implementations of a service from several manufacturers interchangeably. The latter goal makes it possible to partition applications into smaller pieces, possibly implemented and provided by different manufacturers.

The OSGi service gateway shares many of the design goals with personal service environments, and its realization share many properties with the *s*View platform described below. For example, OSGi also provides an open environment in which service components from different manufacturers can be loaded and collaborate to form an application. However, OSGi does not provide *personal* environments. The OSGi service gateway is intended as a service gateway for small groups of users (e.g. a family). The service gateway environment of OSGi is also stationary, which is the natural choice for gateway software. A personal service environment needs to be mobile, since this enables services to follow the user in the network and still execute close to the user and independently of a network connection.

4.4. MExE

The Mobile Station Application Execution Environment (MExE) initiative is a budding standard for platform independent development of services, targeting mobile devices [18, 19]. The initiative includes a classification of mobile devices, which describe minimum levels of capabilities for certain categories of devices. The initiative also adopts W3C's CC/PP protocol [20] for runtime capability and content negotiation of mobile station capabilities and user preferences. This provides a

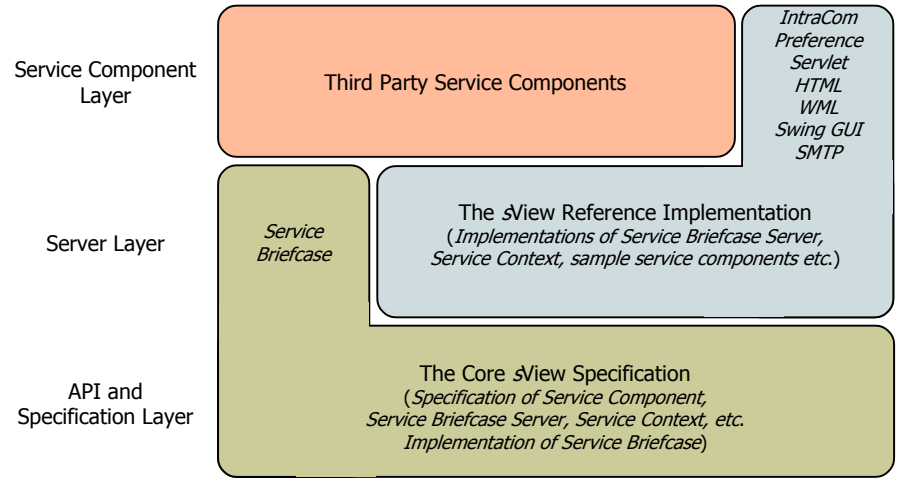

Fig. 1. A schematic overview of the *s*View system and its different parts.

foundation for performing personalization of services and adaptation of content towards mobile devices with different capabilities.

The MExE initiative shares the requirements on device independence and personalization with the personal service environment concept. The crucial distinction between the two approaches lies in how mobility is defined. In the MExE case, mobility means access to electronic services via mobile devices. In the perspective put forth in this paper however, mobility means that the service environments themselves are mobile, and free to migrate (in complete) between hosts.

5. The *s*View System

We have designed and developed *s*View to meet the requirements put forward in Section 2. In *s*View, personal service environments are composed both of services that are mobile and follow the user, and of services that are platform or location specific. In this way, *s*View provides personal service environments that are tailored both to the user and to the usage context.

At a high level, the *s*View system is separated into two parts. The *core sView specification* provides an API for developers of service components and service infrastructure that builds on sView technology. The *sView reference implementation* provides developers with a development and runtime environment for service components as well as a sample implementation of an *s*View server. An illustration of how the core specification and the reference implementation are organized in three layers can be seen in **Fig. 1**.

The architecture and design of the sView system is further described in [21]. Both the core specification and the reference implementation is available for download from http://sview.sics.se.

Mobile / Persistent	Yes	No
Yes	Follows the user and preserves its state (e.g. a calendar).	Does not follow the user but preserves its state (e.g. a printer queue).
No	Follows the user but does not preserve its state (e.g. a proxy to a Web based service).	Does not follow the user nor preserve its state (e.g. a driver for a loudspeaker).

Table 1. Examples of four types of service components.

5.1. The Core *s*View Specification

The core *s*View specification constitutes an API that defines the basics of service components and personal service environment handling. It specifies service components, a runtime environment for service components, a data structure for persistent and mobile images of service environments (service briefcases), and a server for handling service briefcases. The specification has been implemented as a set of Java packages (which contains mostly interfaces and a few classes).

The Service Component. A service provider needs to implement a service component that can be loaded into the users' service environments. The core *s*View specification includes a Java interface that service components must implement. The service component interface includes methods for initializing, starting, suspending, resuming, and stopping the service component. Service components can implement an arbitrarily large part of the functionality of the service, and range from mere proxies to Web based services, to standalone services.

Service components can be declared to be persistent and/or mobile. A persistent service component can save its state together with the service environment when the environment is saved locally on a host or migrates to another host in the network. If a service component is mobile it will follow the service environment as it migrates to a different host in the network. Note that the persistent and mobile properties are orthogonal. A persistent service component that is not mobile can save its state locally, but not migrate to a different host. A mobile service component cannot save its state, neither locally nor while moving; every time such a service component is restarted it will start from scratch (which is fine for many services). The most powerful service component however combines the two properties. Such a service component can both save its state and migrate. **Table 1** lists and exemplifies the four possible combinations of the two properties.

The Service Context. The service context is the entity that maintains and provides runtime support to service components. It manifests the personal service environment in *s*View and is responsible for creating, loading, and removing service components. Once the service components are loaded, the service context controls and sets the

states of the service components. For example, newly created service components should be taken through an initialization state, and when the service environment is about to migrate all service components should be suspended.

When a service environment resumes after having been suspended (e.g. after migrating to a new host), the service context loads service components and other data from a service briefcase. Likewise, when an environment is about to migrate to another host or save its state to disk, service components are stored in a service briefcase.

Service components within a service environment can communicate via service interfaces. A service component that wishes to offer its services to other service components should come with a class that implements an interface to the service. The service context is responsible for mediating service interfaces between service components. It is straightforward to implement a message passing service component on top of the core *s*View service-service communication scheme, and the *s*View reference implementation includes such a service.

Via the service context, service components can control the behavior of the service environment (e.g. migration and shutdown) as well as the behavior of other service components (e.g. creation, suspension, resumption). However, since these activities are sensitive matters, the user must grant privileges to each service component in order for them to perform these actions. A service component may for example be granted the privileges to create and load, but not suspend or kill, service components within the environment. Another service component may be allowed to initiate a migration of the whole service environment to another host, but not to control any aspect of individual service components.

The Service Briefcase. Service briefcases are persistent and mobile images of personal service environments of individual users. The service briefcase is what is actually saved when the user suspends the execution of a service environment to move or shut it down. This is also what is sent between hosts in the network when the user switches interaction devices.

A service briefcase consists of three parts: serialized service components, service component specifications, and preferences.

The Service Briefcase Server. A service briefcase server specifies an API for service briefcase handling. It includes basic functionality such as create new, get, put, and delete briefcase. It also includes functionality for synchronization of content in different instances of a service briefcase on different servers.

Synchronization is used when a service briefcase is to be moved to a server on which it has already been stored. In this case it is possible that parts of the briefcase (e.g. service component specifications) need not be sent with the briefcase. Synchronization is performed in two steps: the first step is to find out the difference between the two service briefcases followed by the second step to update the parts that have been changed. The two-phase commit protocol is used to ensure data integrity during synchronization.

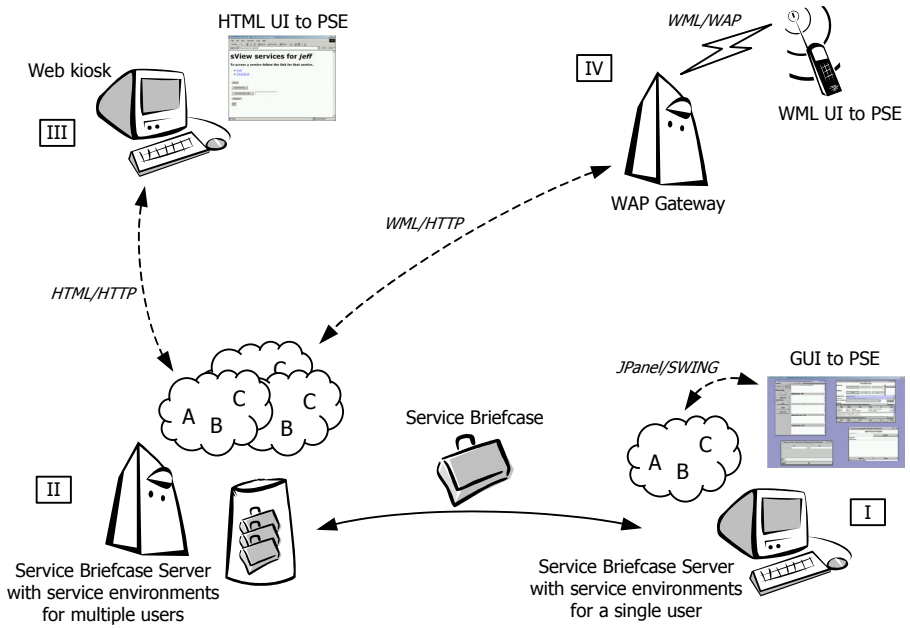

Fig. 2. The main parts of the core sView specification and their relations.

5.2. Dynamic Behavior

This Section describes how services are handled by the *s*View platform. An illustration of the different parts of the core *s*View specification and their relations is given in **Fig. 2**. On the computer marked I a briefcase server and a service environment is executing. In this case the user is sitting next to the same computer as the service environment (represented by the cloud together with service components A, B, and C) is executing on. This makes it possible to use a standard GUI for user-service interaction. The computer marked II hosts service briefcases and environments for several users, which use remote interfaces. One user is using a Web-kiosk with a Web browser for user-service interaction (III) and another user uses a WAP phone (IV). Stored service environments, in the form of service briefcases (illustrated between I and II), can migrate between any computers that run a briefcase server.

Searching for and Adding Service Components. The core *s*View specification does not include any predefined schema for how a user should search for and add service components. Instead, these tasks are left to service components. Every service component within an environment, if created with the correct privileges, is allowed to create and load new service components to the system. Based on this, numerous types of search engines can be implemented.

Saving and Moving the Personal Service Environment. A user that needs to temporarily suspend the execution of a service environment, or move to a different host, has two options. Firstly, if the service environment is executing locally and the user does not intend to move to a different host, the user can save the environment to persistent media on the host. In this case all persistent service components (see Section 5.1) are offered to save their state together with the environment. Secondly, if a network connection is available, the user can move the service environment to a server for remote storage or execution. In this case all mobile service components will migrate with the environment. Service components that in addition are declared as persistent are also offered to save their state with the service environment during the migration. The environment can also migrate directly between two clients, in which case the receiver client will act as a server for remote execution during migration.

The service context includes primitives that allow service components to initiate a save or a migration of a service environment. A service component can be implemented that e.g. automatically saves the environment whenever the user has been idle for more than a few minutes, or actively moves the environment if the user is sighted on a different host.

5.3. The *s*View Reference Implementation

The *s*View reference implementation implements most of the core *s*View specification. It includes a server that implements the service context API (to multiple simultaneous users) and the service briefcase server interface. The server can execute on any computer that hosts a Java 1.3 VM.

The core *s*View specification does not include UI handling. This is instead left as a task for service components to handle. The reference implementation includes service components that handle user interfaces of three types: GUIs specified in Java Swing as well as HTML and WML based user interfaces.

The reference implementation also includes a set of service components for handling other miscellaneous functionality. The *IntraCom (Intra Communication) manager* let service components register a mailbox to which other service components can post messages. The *Preference manager* offers rudimentary handling of preference entries (key and value pairs) for the user. Service components can store and fetch entries, as well as subscribe to changes in the preference database. The user can inspect the database, and control which services should be allowed access to which entries.

5.4. Lessons Learned

The *s*View design and implementation realizes personal service environments, and fulfils the requirements on an open infrastructure for personal mobile services discussed in Section 2. In contrast to related technologies (see Section 4), the use of personal service environments addresses a number of challenges at the same time; such as network independence, control over personal information and the use of different types of devices and user interfaces. The mere work with designing,

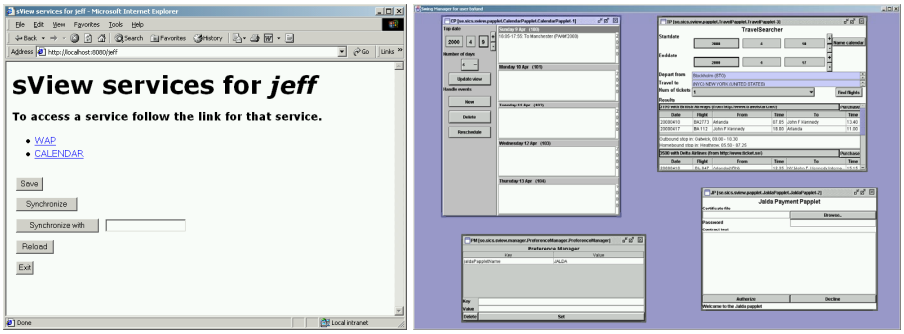

Fig. 3. Screen dumps showing two views of an sView service briefcase: one through an HTML Web browser interface and one through a Java Swing GUI.

implementing and using the *s*View system has shown that the personal service environment approach is feasible.

As always when designing software that is intended for reuse, there is a conflict between making the software general enough to allow for wide usage, and making it specific enough to actually deliver a useful functionality. This conflict turned out to be the biggest challenge when designing the *s*View system. A large effort has been made in choosing suitable levels of abstraction for different parts of the system, in order to make *s*View modular and extensible, yet easy to develop for. An informal evaluation [22] of the feasibility of developing electronic services for *s*View indicated that the current design indeed has succeeded to meet this requirement. However, much wider usage is needed in order to prove whether that conclusion is correct.

6. Experiences with the *s*View System

The research group at SICS makes extensive use of the *s*View system as a base for continued research on electronic services. Our first efforts along this line were to equip the reference implementation with sample services, which illustrate service collaboration and service mobility [22] (see **Fig. 3**). The *Calendar service component* maintains a set of calendar entries for its user. This service offers its user three different interfaces (via the Swing manager, HTML manager, and WML manager). The calendar service also publishes a service interface, so that other service components can add entries to the calendar and check for conflicts between an entry and the user's calendar. The *IMAP service component* implements an e-mail client to IMAP (Internet Message Access Protocol) e-mail accounts. This service offers its user interfaces via both the Swing manager and the WML manager. A demonstrational travel booking service and a payment service (based on the Jalda standard [23]) have also been implemented. The reader should note that the example scenario in Section 3.1 is completely implemented by this set of services.

6.1. Service Designer

A generic problem with service communication is that it is difficult for services to collaborate unless they are explicitly designed to do so, or at least share ontological information.

In the Service Designer project [24], Fredrik Espinoza and et al. use the *s*View system and the SOAP standard [8] in order to explore how end-users themselves can create GUIs to services with only programmatic descriptions of the service available. They have developed a Service Designer service component, which allows users to download descriptions of net-based services, and provides simple means to create a GUI for the services. Once the GUI is finished, a new *s*View service component is compiled based on the net-based service and the GUI.

The service designer also allows the user to combine functionality from several net-based services in one GUI, thus creating explicit collaboration between services that have not been designed to communicate with each other.

Two of the design considerations for *s*View proved essential to this project. Firstly, the open design of *s*View enables the service generator to both support UI design, and the actual generation and installation of the created service. Secondly, the fact that *s*View provides a common locale for service logics makes it possible to create a service that actually inspects service code (in the form of SOAP objects), presents the result of the inspection to the user, and allows users to combine service functionality and create UI's.

6.2. Universal Device Access

The *s*View system promotes use of services from many different types of devices and interfaces. However, the basic *s*View implementation requires that service components are aware of, and designed for, each of the device and interface types that should be used for interaction. If a new device or interface type shows up, existing service components need to be modified.

In the Universal Device Access project [25] we have designed a method that simplifies the process of developing services that can be accessed from different user interface types. The method allows both service- and user-driven interaction, unlike UIML-based [26] applications or Web-style-interaction that are entirely user-driven.

In analogy with HTML [7] and similar script languages, we separate service logic and user interface with a general language for specifying interaction. Different devices can then implement this script language in a way that is specific for the device and interface type.

In this project, the open nature of *s*View has proven useful, in particular to enable services to fully or partly override this functionality. If the service component wishes to tailor the user interface on a particular device it can send a customization form to the interaction interpreter. The interpreter will then replace its standard interpretation of the interaction language by using this form to render output and interpret input.

6.3. GeoNotes – Virtual Notes in the Real World

GeoNotes [27-29] is an *s*View service component that lets its user annotate physical locations with virtual notes. Such a note is intended to say something about the location, just as a Post-it note can express something about the object it is attached to. Other users of the same service get the notes as they pass by an annotated location.

The GeoNotes service relies heavily on mobile and context-sensitive usage. Developing GeoNotes as an *s*View service component has therefore been an important test of how suitable *s*View is for mobile and context dependent services. *s*View provided good support for both of these requirements [28]. In particular, *s*View proved useful for separating personal and common information in a way that protects user privacy: the GeoNotes server is restricted to holding common information about the posted notes, whereas the personal information is kept within the user's (mobile) personal service environment.

7. Conclusions

We have shown that the personal service environment concept provides a powerful tool for handling electronic services. The key issues are openness and user control. Systems based on personal service environments support adding and removing services and users without affecting others. Furthermore, such systems are focused on creating a personal space that the user has full control over.

The *s*View system is a fully functional implementation that is based on the personal service environment concept, as well as a specification of how to implement services, which proves that the concept is realistically feasible. Even though the *s*View system is only a prototype intended as a research tool, we are satisfied with its performance and we have used it for internal development in several projects [24, 25, 27-29].

Our solution is deliberately thin and generic tools are needed in order to make it accessible and useful. However, the core *s*View specification is flexible enough to allow for almost unlimited development of such extensions.

8. Acknowledgements

The work presented in this paper has been funded by The Swedish Institute for Information Technology (www.siti.se). Thanks to the members of the HUMLE laboratory at the Swedish Institute of Computer Science (www.sics.se/humle), in particular Fredrik Espinoza, for inspiration and thoughtful comments.

References

[1] A. Dearle, "Toward Ubiquitous Environments for Mobile Users," *IEEE Internet Computing*, vol. 2, pp. 22-32, 1998.

[2] L. Kleinrock, "Nomadicity anytime, anywhere in a disconnected world," *Mobile Networks and Applications*, vol. 1, pp. 351-357, 1996.

[3] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," *ACM Transactions on Computer-Human Interaction*, vol. 7, pp. 3-28, 2000.

[4] J. S. Hansen, T. Reich, B. Andersen, and E. Jul, "Dynamic Adaptation of Network Connections in Mobile Environments," *IEEE Internet Computing*, vol. 2, pp. 39-47, 1998.

[5] N. Davies, A. Friday, G. S. Blair, and K. Cheverst, "Distributed Systems Support for Adaptive Mobile Applications," *Mobile Networks and Applications*, vol. 1, 1996.

[6] E. Volokh, "Personalization and Privacy," *Communications of the ACM*, vol. 43, pp. 84-88, 2000.

[7] T. Berners-Lee, R. Caillau, J.-F. Groff, and B. Pollermann, "World-Wide Web: The Information Universe," *Electronic Networking: Research, Applications and Policy*, vol. 2, pp. 52-58, 1992.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," World Wide Web Consortium, W3C Note 27 July 1999, May 8, 2000.

[9] A. Di Stefano and C. Santoro, "NetChaser: Agent Support for Personal Mobility," *IEEE Internet Computing*, vol. 4, pp. 74-79, 2000.

[10] J. E. White, "Mobile Agents," in *Software Agents*, J. M. Bradshaw, Ed. Menlo Park, CA: AAAI Press/MIT Press, ISBN 0-262-52234-9, 1997, pp. 437-472.

[11] A. Waern, M. Tierney, Å. Rudström, and J. Laaksolahti, "ConCall: An information service for researchers based on EdInfo," Swedish Institute of Computer Science, Kista, T98-04, 1998.

[12] M. Tierney, "ConCall: An Exercise in Designing Open Service Architectures," Ph.Lic. thesis, The Royal Institute of Technology and Stockholm University, Stockholm, Sweden, 2000.

[13] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes, "Hive: Distributed Agents for Networking Things," presented at First International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents featuring the Third Dartmouth Workshop on Transportable Agents, Rancho Las Palmas Marriott's Resort and Spa, Palm Springs, CA, 1999.

[14] C. Pullela, L. Xu, D. Chakraborty, and A. Joshi, "A Component Based Architecture for Mobile Information Access," Department of Computing Science and Electrical Engineering, University of Maryland Baltimore County, Technical Report, TR-CS-00-05, March 31, 2000.

[15] H. L. Chen, "Developing a Dynamic Distributed Intelligent Agent Framework Based on the Jini Architecture," M.Sc. thesis, University of Maryland Baltimore County, Baltimore, 2000.

[16] T. Sandholm and Q. Huai, "Nomad: Mobile Agent System for an Internet-Based Auction House," *IEEE Computer*, vol. 4, pp. 80-86, 2000.

[17] "OSGi Service Gateway Specification Release 1.0," Open Services Gateway Initiative, May, 2000.

[18] "Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Mobile Station Application Execution Environment (MExE); Functional description; Stage 2," European Telecommunications Standards Institute, ETSI TS 123 057 v.3.0.0, January, 2000.

[19] "Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Mobile Station Application Execution Environment (MExE); Service description; Stage 1," European Telecommunications Standards Institute, ETSI TS 122 057 v.3.0.1, January, 2000.

[20] F. Reynolds, J. Hjelm, S. Dawkins, and S. Singhal, "Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation," World Wide Web Consortium, W3C Note 27 July 1999, July 27, 1999.

[21] M. Bylund, "*s*View - Architecture Overview and System Description," Swedish Institute of Computer Science, Kista, Sweden, SICS Technical Report T2001:06, May, 2001.

[22] M. Boman, "Implementing services for a PSE," M.Sc. thesis, Uppsala University, Uppsala, Sweden, 2000.

[23] "The Jalda Charging API, Release 1.3," EHPT Sweden AB, available at: http://www.jalda.com/ [2001, April 18], 2000, May.

[24] F. Espinoza and O. Hamfors. "ServiceDesigner: Enabling End-Users Access to Web Services," Unpublished manuscript, available at: http://sview.sics.se, 2001.

[25] S. Nylander and M. Bylund. "Providing Universal Device Access to Mobile Services," Unpublished manuscript, available at: http://sview.sics.se, 2001.

[26] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "UIML: an Appliance-Independent XML User Interface Language," *Computer Networks*, vol. 31, pp. 1695-1708, 1999.

[27] F. Espinoza, P. Persson, A. Sandin, H. Nyström, E. Cacciatore, and M. Bylund, "GeoNotes: Social Filtering of Position-Based Information," Swedish Institute of Computer Science, SICS Technical Report T2001:08, May, 2001.

[28] H. Nyström and A. Sandin, "Social Mobile Services in an Open Service Environment - an Overview, Analysis and Implementation," M.Sc. thesis, Uppsala University, Uppsala, Sweden, 2001.

[29] P. Persson, F. Espinoza, and E. Cacciatore, "GeoNotes: Social Enhancement of Physical Space," presented at CHI'2001, Seattle, WA, 2001.